第 3 章　外设驱动控制

ESP32-C3 和外围设备构成了感知控制层，感知控制层位于物联网系统结构的第一层，极其重要。外围设备包括传感模块和执行模块。

- 传感模块：包括温湿度/光敏/压力传感器、按键、限位开关和触摸屏等，负责感知和获取外界的信息并转换成 ESP32-C3 能接收的数字信号。
- 执行模块：包括继电器、电机、LED、显示屏等，负责接收 ESP32-C3 传递过来的数字信号并执行对应的指令动作。

ESP32-C3 通过各种通信接口跟外围设备进行交互，ESP32-C3 拥有 22 个 I/O，支持 UART、SPI、I^2C、I^2S、PWM、ADC、TWA 等多种通信接口，满足各种应用场景及复杂的应用。

本章主要介绍 ESP32-C3 的 GPIO、ADC、UART、I^2C、SPI 等接口的基础知识和常用函数，并编写程序使用这 5 个接口驱动和控制外围设备。

3.1　GPIO 应用

本节介绍 GPIO 的基础知识及其常用函数，并通过两个动手实践项目帮助读者掌握 GPIO 的相关知识点。

3.1.1　GPIO 简介

GPIO（General-Purpose Input/Output，通用型输入、输出接口），是嵌入式系统中最常用的接口。GPIO 可以读取外部设备的电平状态（高电平或者低电平），输出高电平或者低电平控制外部设备，还可以通过控制时序来软件模拟 UART、SPI、I^2C 等通用接口。

不同型号的 ESP32 芯片具有不同的 GPIO 管脚数，如表 3.1 所示，每个 GPIO 管脚都可以作为一个通用 I/O，通过 GPIO 交换矩阵和 I/O MUX，可配置任何一个 GPIO 管脚。

表 3.1　ESP32 系列芯片的GPIO管脚

芯　片	管 脚 数	管 脚 号
ESP32	34	GPIO0~GPIO19、GPIO21~GPIO23、GPIO25~GPIO27和GPIO32~GPIO39
ESP32-S2	43	GPIO0~GPIO21和GPIO26~GPIO46
ESP32-S3	45	GPIO0~GPIO21和GPIO26~GPIO48
ESP32-C2	21	GPIO0~GPIO20
ESP32-C3	22	GPIO0~GPIO21

续表

芯　　片	管　脚　数	管　脚　号
ESP32-C6	31	GPIO0~GPIO30
ESP32-H2	28	GPIO0~GPIO27
ESP32-P4	55	GPIO0~GPIO54

3.1.2　GPIO 的常用函数

GPIO 的常用函数如表 3.2 所示，其中，gpio_config()是使用 GPIO 实现输入/输出功能不可或缺的核心配置函数，该函数的入参和返回值如下：

```
/**
 * @brief 配置 GPIO 参数。
 *
 * @param[gpio_config_t *] pGPIOConfig: GPIO 配置参数，该结构体见代码 3.1。
 *
 * @return
 *      - ESP_OK，成功。
 *      - ESP_FAIL，失败。
 */
esp_err_t gpio_config(const gpio_config_t *pGPIOConfig);
```

表 3.2　GPIO的常用函数

属性/函数	说　　明
gpio_config()	配置GPIO的参数（模式、上拉、下拉和中断类型等）
gpio_reset_pin()	重置GPIO为默认状态
gpio_set_direction()	设置GPIO的输入、输出方向
gpio_set_pull_mode()	设置GPIO的上拉、下拉模式
gpio_set_intr_type()	设置GPIO的中断触发类型
gpio_intr_enable()	启用GPIO中断服务
Gpio_intr_disable()	禁用GPIO中断服务
gpio_install_isr_service()	安装GPIO中断服务
gpio_isr_handler_add()	添加GPIO中断事件处理函数
gpio_isr_handler_remove()	移除GPIO中断事件处理函数
gpio_set_level()	GPIO输出高低电平
gpio_get_level()	GPIO读取高低电平

代码 3.1　GPIO配置结构体

```
/**
 * 说明：GPIO 配置参数
 */
typedef struct {
    uint64_t pin_bit_mask;              /*!< GPIO 引脚选择：按位映射        */
    gpio_mode_t mode;                   /*!< GPIO 模式：设置输入/输出模式   */
    gpio_pullup_t pull_up_en;           /*!< GPIO 上拉使能                  */
    gpio_pulldown_t pull_down_en;       /*!< GPIO 下拉使能                  */
    gpio_int_type_t intr_type;          /*!< GPIO 中断类型                  */
} gpio_config_t;
```

3.1.3　实践：通过 GPIO 监听按键

【ESP32 源码路径：tutorial-esp32c3-getting-started/tree/master/peripheral/button】

本节主要利用前面介绍的 GPIO 知识点和 GPIO 的常用函数进行实践：通过 GPIO 实现监听按键被按下的功能。

1．操作步骤

（1）准备一个 ESP32-C3 开发板，按照表 3.3 所示的按键接线说明做好硬件准备。

（2）通过 Visual Studio Code 开发工具编译 button 工程源码，生成相应的固件，再将固件下载到 ESP32-C3 开发板上。

（3）ESP32-C3 运行程序后，按 BOOT 按键或者放开 BOOT 按键，会打印当前触发的 GPIO 引脚号和该引脚当前的电平状态，程序运行日志如图 3.1 所示。

```
I (197) cpu_start: ESP-IDF:          v5.1.2-dirty
I (203) cpu_start: Min chip rev:     v0.3
I (207) cpu_start: Max chip rev:     v0.99
I (212) cpu_start: Chip rev:         v0.3
I (217) heap_init: Initializing. RAM available for dynamic allocation:
I (224) heap_init: At 3FC8CA50 len 000335B0 (205 KiB): DRAM
I (230) heap_init: At 3FCC0000 len 0001C710 (113 KiB): DRAM/RETENTION
I (238) heap_init: At 3FCDC710 len 00002950 (10 KiB): DRAM/RETENTION/STACK
I (245) heap_init: At 50000010 len 00001FD8 (7 KiB): RTCRAM
I (252) spi_flash: detected chip: generic
I (256) spi_flash: flash io: dio
I (260) sleep: Configure to isolate all GPIO pins in sleep state
I (267) sleep: Enable automatic switching of GPIO sleep configuration
I (274) app_start: Starting scheduler on CPU0
I (00:00:00.107) main_task: Started on CPU0
I (00:00:00.112) main_task: Calling app_main()
I (00:00:00.117) gpio: GPIO[8]| InputEn: 1| OutputEn: 0| OpenDrain: 0| Pullup: 1| Pulldown: 0| Intr:3
I (00:00:00.127) gpio: GPIO[9]| InputEn: 1| OutputEn: 0| OpenDrain: 0| Pullup: 1| Pulldown: 0| Intr:3
I (00:00:01.007) button: GPIO[9] 中断触发，当前电平: 0
I (00:00:01.157) button: GPIO[9] 中断触发，当前电平: 1
```

图 3.1　实现按键功能的程序运行日志

2．按键介绍和接线说明

ESP32-C3-DevKitM-1 开发板上有两个按键，即 RESET 按键和 BOOT 按键。RESET 按键用于硬件复位，BOOT 按键用于启动模式切换（不按 BOOT 按键复位，ESP32-C3 正常启动；按住 BOOT 按键再复位，ESP32-C3 进入固件下载模式）。ESP32-C3 正常启动后，GPIO9 可以作为普通的 GPIO 使用，BOOT 按键可以作为普通按键使用。除此之外，我们再外接一组按键到 GPIO8 上。

BOOT 按键原理如图 3.2 所示，SW1 是 BOOT 按键，SW2 是 RESET 按键。SW1 的 1 脚接 GND，SW1 的 3 脚接 GPIO9，当 BOOT 按键未按下时，GPIO9 悬空高电平。当 BOOT 按键被按下时，SW1 的 1 和 3 脚短接，GPIO9 短接 GND 低电平。

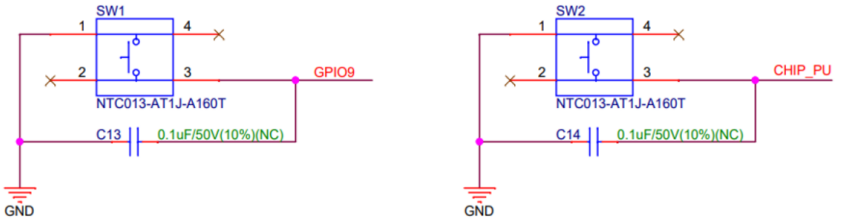

图 3.2　BOOT 按键原理

按键接线说明如表 3.3 所示，两个按键，板载 BOOT 按键连接 GPIO9，外部按键模块连接 GPIO8。

表 3.3　按键接线说明

外围设备引脚	ESP32-C3 开发板引脚	说　　明
VCC	VCC	电源正极
GND	GND	电源负极（地）
板载按键BOOT，SW1-3引脚	GPIO9	ESP32-C3 GPIO输入
外部按键模块	GPIO8	ESP32-C3 GPIO输入

3．程序源码解析

程序源码从 app_main()函数开始，首先完成 GPIO 初始化并创建一个名为 button_queue 的队列。关于任务和队列的知识点在第 4 章详细介绍，本次实践的要点在于 GPIO 输入初始化和按键功能的实现。

GPIO 输入初始化函数见代码 3.2，使用 gpio_config()配置 GPIO 输入参数，使用 gpio_isr_handler_add()函数添加中断事件处理程序。GPIO 配置结构体见代码 3.1，其中包含引脚选择、引脚模式（输入/输出）、上拉使能、下拉使能、中断触发类型（上升沿/下降沿）等配置。

代码 3.2　GPIO初始化

```
// 宏定义 GPIO 输入的引脚号
#define GPIO_INPUT_BOOT         9
#define GPIO_INPUT_EXTER        8
#define GPIO_INPUT_PIN_SEL  ((1ULL<<GPIO_INPUT_BOOT) | (1ULL<<GPIO_INPUT_
EXTER))
#define ESP_INTR_FLAG_DEFAULT  0

/**
 * 说明：GPIO 初始化
 */
void gpio_init(void)
{
    gpio_config_t io_conf = {};
    //设置 I/O 引脚：在宏定义中修改
    io_conf.pin_bit_mask = GPIO_INPUT_PIN_SEL;
    //设置 I/O 方向：输入
    io_conf.mode = GPIO_MODE_INPUT;
    //设置 I/O 上拉：使能
    io_conf.pull_up_en = 1;
    //设置 I/O 中断类型：上升沿中断和下降沿
    io_conf.intr_type = GPIO_INTR_ANYEDGE;
    //配置 GPIO 参数
    gpio_config(&io_conf);

    //安装 GPIO 中断服务
    gpio_install_isr_service(ESP_INTR_FLAG_DEFAULT);
    //添加 GPIO_INPUT_BOOT 中断事件处理函数
    gpio_isr_handler_add(GPIO_INPUT_BOOT, gpio_isr_handler, (void*)
GPIO_INPUT_BOOT);
    //添加 GPIO_INPUT_EXTER 中断事件处理函数
```

```
    gpio_isr_handler_add(GPIO_INPUT_EXTER, gpio_isr_handler, (void*)
GPIO_INPUT_EXTER);
}
```

本次实践的程序流程如图 3.3 所示，初始化完成后程序进入一个循环，堵塞读取 button_queue 队列数据，一旦队列存在数据，就将数据读取出来并打印日志。

当 BOOT 按键和外部按键被按下时，会触发中断并进入中断事件处理函数，在中断事件处理函数中向 button_queue 队列写入当前触发中断的 GPIO 引脚号。这时候堵塞读取 button_queue 队列的任务，就可以将数据打印出来。

注意：在中断事件处理函数中不能做延时和堵塞的操作，并且不能显示日志打印。

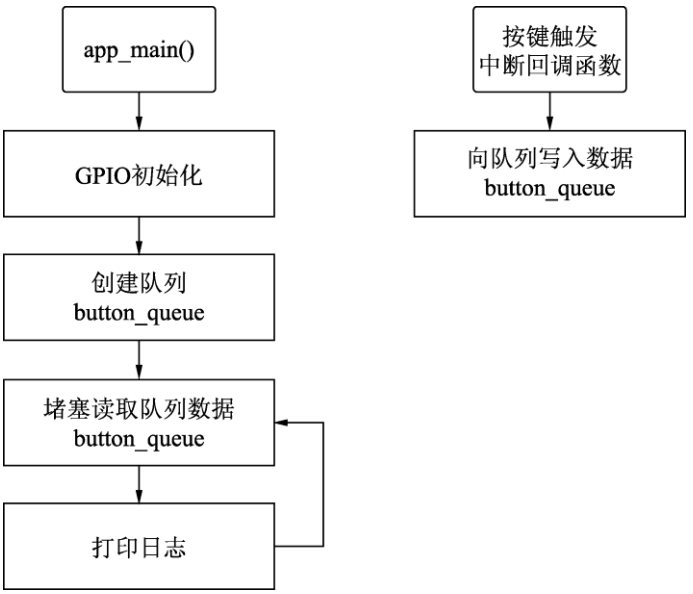

图 3.3　实现按键功能的程序流程

3.1.4　实践：通过 GPIO 控制 LED 亮灭

【ESP32 源码路径：tutorial-esp32c3-getting-started/tree/master/peripheral/led】

本节主要利用前面介绍的 GPIO 知识点和 GPIO 的常用函数进行实践：通过 GPIO 输入实现控制 LED 亮灭的功能。

1．操作步骤

（1）准备一个 ESP32-C3 开发板，按照表 3.4 所示的发光 LED 模块接线说明做好硬件准备。

表 3.4　发光LED模块接线说明

外围设备引脚	ESP32-C3 开发板引脚	说　　明
VCC	VCC	电源正极
GND	GND	电源负极（地）
OUT	GPIO7	ESP32-C3 GPIO输出

（2）通过 Visual Studio Code 开发工具编译 led 工程源码，生成相应的固件，再将固件下载到 ESP32-C3 开发板上。

（3）ESP32-C3 运行程序后，按 BOOT 按键，GPIO 输出高电平控制 LED 点亮。

（4）按外部按键，GPIO 输出低电平控制 LED 熄灭，程序运行日志如图 3.4 所示。

```
I (193) cpu_start: ESP-IDF:          v5.1.2
I (198) cpu_start: Min chip rev:     v0.3
I (202) cpu_start: Max chip rev:     v0.99
I (207) cpu_start: Chip rev:         v0.3
I (212) heap_init: Initializing. RAM available for dynamic allocation:
I (219) heap_init: At 3FC8C9A0 len 00033660 (205 KiB): DRAM
I (226) heap_init: At 3FCC0000 len 0001C710 (113 KiB): DRAM/RETENTION
I (233) heap_init: At 3FCDC710 len 00002950 (10 KiB): DRAM/RETENTION/STACK
I (240) heap_init: At 50000010 len 00001FD8 (7 KiB): RTCRAM
I (247) spi_flash: detected chip: generic
I (251) spi_flash: flash io: dio
I (255) sleep: Configure to isolate all GPIO pins in sleep state
I (262) sleep: Enable automatic switching of GPIO sleep configuration
I (269) app_start: Starting scheduler on CPU0
I (274) main_task: Started on CPU0
I (274) main_task: Calling app_main()
I (274) gpio: GPIO[8]| InputEn: 1| OutputEn: 0| OpenDrain: 0| Pullup: 1| Pulldown: 0| Intr:1
I (284) gpio: GPIO[9]| InputEn: 1| OutputEn: 0| OpenDrain: 0| Pullup: 1| Pulldown: 0| Intr:1
I (294) gpio: GPIO[7]| InputEn: 0| OutputEn: 1| OpenDrain: 0| Pullup: 1| Pulldown: 0| Intr:0
I (3524) led: GPIO[9] intr, val: 1
I (3524) led: led on
I (6314) led: GPIO[8] intr, val: 1
I (6314) led: led off
```

图 3.4　控制 LED 的程序运行日志

2．LED介绍和接线

发光 LED 模块如图 3.5 所示，工作电压为 3.3~5V，工作温度为−40℃~80℃，由发光二极管驱动，高电平点亮，低电平熄灭。

发光 LED 模块接线说明如表 3.4 所示，ESP32-C3 的 GPIO7 作为输出，控制 LED 的亮灭。

3．程序源码解析

本次实践工程的源码结构如图 3.6 所示，包含 led.c、led.h、button.c、button.h 和 main.c 几个源文件。对比上一个实践只有一个源文件，本次实践源码模块解耦分离开，代码更加清晰易懂，提升了代码的扩展性和可读性，方便后续的开发和维护。

图 3.5　发光 LED 模块

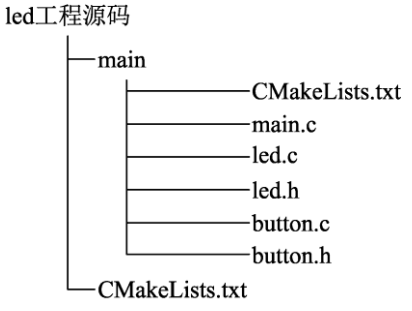

图 3.6　控制 LED 功能的工程源码结构

- main.c：主程序流程如图 3.7 所示，在应用程序的入口 app_main()函数中完成按键初始化和 LED 初始化，然后创建一个名为 button_queue 的队列，最后进入一个循环，堵塞读取 button_queue 队列数据，等待按键按下触发中断向队列写入数据。一旦队列存在数据，就将数据读取出来，转换成 GPIO 引脚号，然后判断如果是 BOOT

按键按下，则 GPIO 输出高电平控制 LED 点亮。当外部按键按下时，GPIO 输出低电平控制 LED 熄灭。具体实现可参考代码 3.3。

- led.c 和 led.h：声明和实现 LED 初始化函数，使用 gpio_config()配置 GPIO 输出参数，具体实现可参考代码 3.4。
- button.c 和 button.h：实现按键初始化函数和按键触发中断事件处理函数，详情可查阅 3.1.3 节。

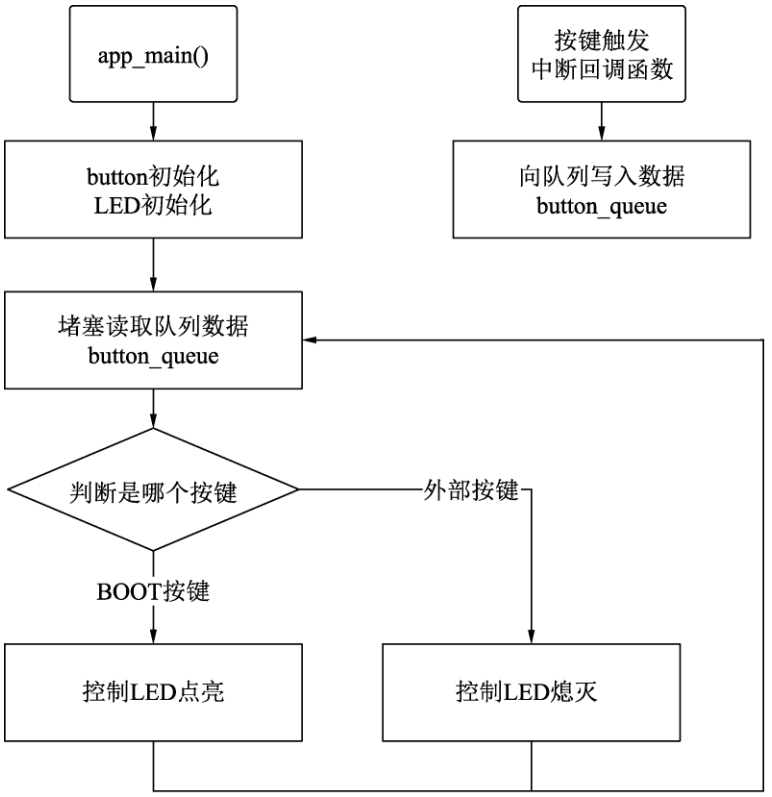

图 3.7　控制 LED 功能的程序流程

代码 3.3　按键控制LED亮灭的关键代码

```
void app_main(void)
{
    uint32_t io_num;

    //按键初始化
    button_init();

    //led 初始化
    led_init();

    while(true) {
        // 堵塞读取队列中的数据
        if(xQueueReceive(button_queue, &io_num, portMAX_DELAY)) {
            // 如果读取成功，则打印出 GPIO 引脚号和当前的电平
            ESP_LOGI(TAG, "GPIO[%ld] 触发中断, 当前电平: %d", io_num,
gpio_get_level(io_num));
```

```
            if(io_num==GPIO_INPUT_BOOT) {
                ESP_LOGI(TAG, "led on");
                gpio_set_level(GPIO_OUT_LED, 1);
            } else if(io_num==GPIO_INPUT_EXTER) {
                ESP_LOGI(TAG, "led off");
                gpio_set_level(GPIO_OUT_LED, 0);
            }
        }
    }
}
```

代码 3.4　LED初始化函数

```
/**
 * @brief led 初始化
 */
void led_init(void)
{
    gpio_config_t io_conf = {0};
    //设置 I/O 引脚：在宏定义中修改
    io_conf.pin_bit_mask = GPIO_OUT_PIN_SEL;
    //设置 I/O 方向：输出
    io_conf.mode = GPIO_MODE_OUTPUT;
    //设置 I/O 上拉：使能
    io_conf.pull_up_en = 1;
    //配置 GPIO 参数
    gpio_config(&io_conf);
}
```

由于添加了 led.c 和 button.c 两个源文件，所以需要在 CMakeLists.txt 配置文件中添加，见代码 3.5，否则 ESP-IDF 在构建系统时不会将 led.c 和 button.c 文件加入编译。

代码 3.5　CMakeLists.txt

```
idf_component_register(SRCS "main.c" "led.c" "button.c"
                    INCLUDE_DIRS ".")
```

3.2　ADC 应用

本节介绍 ADC 的基础知识及其常用函数，并通过一个动手实践项目帮助读者掌握 ADC 读取模拟信号的相关知识点。

3.2.1　ADC 简介

ADC（Analog-to-Digital Converter）是将模拟量转换成数字量的模数转换器。如表 3.5 所示，ESP32-C3 集成了 2 个 12 位 ADC 单元，共支持 6 个通道的模拟信号检测，其主要特性如下：

- 12 位采样分辨率；
- 提供 6 个引脚的模拟电压采集转换；
- 提供两个滤波器，滤波系数可配；
- 提供 DMA，可高效获取 ADC 转换结果；

- ❑ 支持单次转换模式和多通道扫描模式；
- ❑ 支持在多通道扫描模式下，自定义扫描通道顺序；
- ❑ 支持阈值监控，当采样值大于设置的高阈值或小于设置的低阈值时将产生中断。

表 3.5　ESP32-C3 ADC通道

引　　脚	通道编号引脚	ADC选择
GPIO0	0	SAR ADC1
GPIO1	1	
GPIO2	2	
GPIO3	3	
GPIO4	4	
GPIO5	0	SAR ADC2
内部电压	n/a	

ADC 转换模拟信号时，转换分辨率（12 位）电压（mV）范围为 0 ~V_{ref}。其中，V_{ref} 为 ADC 内部参考电压。因此，转换结果 (data) 可以使用以下公式转换成模拟电压输出 V_{data}：

$$V_{data} = \frac{V_{ref}}{4095} \times \text{data}$$

如果需要转换大于 V_{ref} 的电压，则在信号输入 ADC 前可进行衰减。衰减可配置为 0dB、2.5 dB、6 dB 和 12 dB。

最后需要注意的是，ESP32-C3 的 ADC2 无法正常工作，建议使用 ADC1，详见 ESP32-C3 系列芯片勘误表。

3.2.2　ADC 的常用函数

ADC 的常用函数分为 ADC 单次转换类别函数和 ADC 连续转换类别函数，用来满足不同应用场景的需求，如表 3.6 所示。

表 3.6　ADC的常用函数

属性/函数	说　　明
adc_oneshot_new_unit()	创建ADC单次转换句柄
adc_oneshot_config_channel()	设置ADC单次转换模式的参数
adc_oneshot_read()	获取ADC单次转换的原始结果
adc_oneshot_get_calibrated_result()	获取校准结果（mV）的快捷函数。相当于adc_oneshot_read()和adc_cali_raw_to_voltage()
adc_oneshot_del_unit()	删除ADC单次转换的句柄
adc_cali_create_scheme_curve_fitting()	创建ADC校准曲线拟合方案句柄
adc_cali_raw_to_voltage()	将原始结果转换成校准结果（mV）
adc_cali_delete_scheme_curve_fitting()	删除ADC校准曲线拟合方案句柄
adc_continuous_new_handle()	创建ADC连续转换句柄
adc_continuous_config()	设置ADC连续转换模式的参数

续表

属性/函数	说　明
adc_continuous_register_event_callbacks()	ADC连续转换事件注册回调函数
adc_continuous_start()	开始ADC连续转换
adc_continuous_read()	读取ADC连续转换结果
adc_continuous_stop()	停止ADC连续转换
adc_continuous_deinit()	取消ADC连续转换初始化
adc_continuous_flush_pool()	刷新ADC连续转换缓冲区

3.2.3　实践：通过 ADC 读取实现光线强度检测

【ESP32 源码路径：tutorial-esp32c3-getting-started/tree/master/peripheral/adc】

本节主要利用前面介绍的 ADC 知识点和 ADC 的常用函数进行实践：通过 ADC 读取光敏传感器的输出电压，实现光线强度检测。

1．操作步骤

（1）准备一个 ESP32-C3 开发板，按照表 3.7 所示的光敏电阻传感器模块接线说明做好硬件准备。

（2）通过 Visual Studio Code 开发工具编译 adc 工程源码，生成相应的固件，再将固件下载到 ESP32-C3 开发板上。

（3）ESP32-C3 运行程序后，在 ADC 初始化校准之后，就不断读取光敏传感器输出的电压值，这时候拿手机闪光灯持续靠近光敏传感器，光敏传感器受到强光影响，输出的电压值将不断降低，程序运行日志如图 3.8 所示。

```
I (203) cpu_start: ESP-IDF:          v5.1.2
I (207) cpu_start: Min chip rev:     v0.3
I (212) cpu_start: Max chip rev:     v0.99
I (217) cpu_start: Chip rev:         v0.3
I (222) heap_init: Initializing. RAM available for dynamic allocation:
I (229) heap_init: At 3FC8C840 len 000337C0 (205 KiB): DRAM
I (235) heap_init: At 3FCC0000 len 0001C710 (113 KiB): DRAM/RETENTION
I (242) heap_init: At 3FCDC710 len 00002950 (10 KiB): DRAM/RETENTION/STACK
I (250) heap_init: At 50000014 len 00001FD4 (7 KiB): RTCRAM
I (257) spi_flash: detected chip: generic
I (261) spi_flash: flash io: dio
I (265) sleep: Configure to isolate all GPIO pins in sleep state
I (271) sleep: Enable automatic switching of GPIO sleep configuration
I (279) app_start: Starting scheduler on CPU0
I (284) main_task: Started on CPU0
I (284) main_task: Calling app_main()
I (284) gpio: GPIO[2]| InputEn: 0| OutputEn: 0| OpenDrain: 0| Pullup: 0| Pulldown: 0| Intr:0
I (294) ADC: Calibration Success
I (304) ADC: ADC1 Channel[2] Raw Data: 4014, Cali Voltage: 2892 mV
I (1304) ADC: ADC1 Channel[2] Raw Data: 4095, Cali Voltage: 2942 mV
I (2304) ADC: ADC1 Channel[2] Raw Data: 3727, Cali Voltage: 2708 mV
I (3304) ADC: ADC1 Channel[2] Raw Data: 1421, Cali Voltage: 1062 mV
I (4304) ADC: ADC1 Channel[2] Raw Data: 991, Cali Voltage: 744 mV
I (5304) ADC: ADC1 Channel[2] Raw Data: 477, Cali Voltage: 361 mV
I (6304) ADC: ADC1 Channel[2] Raw Data: 288, Cali Voltage: 219 mV
I (7304) ADC: ADC1 Channel[2] Raw Data: 1454, Cali Voltage: 1087 mV
I (8304) ADC: ADC1 Channel[2] Raw Data: 153, Cali Voltage: 117 mV
```

图 3.8　通过 ADC 读取实现光线强度检测的程序运行日志

2．光敏传感器介绍和接线

光敏电阻传感器模块如图 3.9 所示，其对环境光线最敏感，一般用来检测周围环境光

线的亮度。光敏电阻传感器模块的工作电压为 3.3~5V，板载电位器可以调整光线检测的阈值。

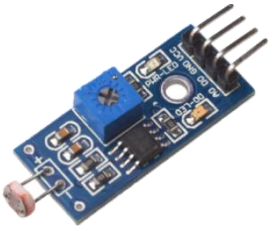

图 3.9　光敏电阻传感器模块

模块输出有两种形式：DO（Digital Output，数字输出信号）输出高电平或者低电平，当环境光线亮度超过阈值时，DO 输出低电平，否则输出高电平；AO（Analog Output，模拟输出信号）输出电压值，该电压值在一定程度上反映了当前的环境光线强度，如果想要得到具体的转换公式，可以通过测量不同光强（发光强度）下的电压值数据，利用数学建模的方法，推导出电压到光强的准确公式。

光敏电阻传感器模块接线说明如表 3.7 所示，光敏电阻传感器模块的 DO 悬空不接，AO 接 ESP32-C3 开发板的 GPIO0 做 ADC 采集。

表 3.7　光敏电阻传感器模块接线说明

外围设备引脚	ESP32-C3 开发板引脚	说　　明
VCC	VCC	电源正极3.3~5V
GND	GND	电源负极（地）
DO	×	数字信号（高电平或者低电平）
AO	GPIO2	模拟信号（电压值）

3．程序源码解析

本次实践的程序流程如图 3.10 所示，使用 ADC 单次转换模式，首先完成总共 3 个步骤的 ADC 单次转换初始化操作，最后程序进入一个循环，间隔 1s 循环读取 ADC 转换的原始数据，再将原始数据转换成校准电压数据（单位为 mV，毫伏）。

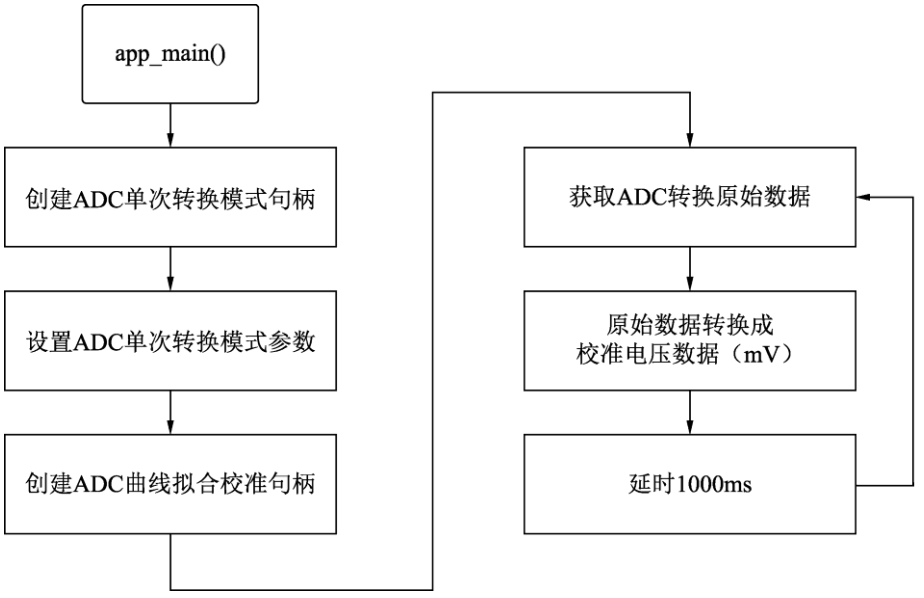

图 3.10　通过 ADC 读取实现光线强度检测的程序流程

ADC 单次转换初始化关键代码如代码 3.6 所示。ADC 单次转换初始化操作包含创建 ADC 单次转换模式句柄、设置 ADC 单次转换模式参数和创建 ADC 曲线拟合校准句柄，

主要工作包括选定 ADC 单元和通道、设定 ADC 转换结果的位宽以及配置 ADC 的衰减参数。

代码 3.6　ADC单次转换初始化关键代码

```
// ADC 单次转换模式句柄
adc_oneshot_unit_handle_t adc1_handle;
// ADC 单次转换模式驱动初始化参数
adc_oneshot_unit_init_cfg_t init_config1 = {
    // ADC 单元
    .unit_id = ADC_UNIT_1,
};
// 创建 ADC 单次转换模式句柄
adc_oneshot_new_unit(&init_config1, &adc1_handle);

// ADC 通道配置参数
adc_oneshot_chan_cfg_t config = {
    // ADC 转换结果位宽
    .bitwidth = ADC_BITWIDTH_DEFAULT,
    // ADC 衰减
    .atten = ADC_ATTEN_DB_11,
};
// 设置 ADC 单次转换模式的参数
adc_oneshot_config_channel(adc1_handle, ADC_CHANNEL_2, &config);

// ADC 校准句柄
adc_cali_handle_t adc1_cali_handle = NULL;
// ADC 曲线拟合校准方案配置参数
adc_cali_curve_fitting_config_t cali_config = {
    // ADC 单元
    .unit_id = ADC_UNIT_1,
    // ADC 通道
    .chan = ADC_CHANNEL_2,
    // ADC 衰减
    .atten = ADC_ATTEN_DB_11,
    // ADC 原始数据输出位宽
    .bitwidth = ADC_BITWIDTH_DEFAULT,
};
// 创建 ADC 曲线拟合校准方案句柄
esp_err_t ret = adc_cali_create_scheme_curve_fitting(&cali_config,
&adc1_cali_handle);
if (ret == ESP_OK) {
    ESP_LOGI(TAG, "Calibration Success");
} else if (ret == ESP_ERR_NOT_SUPPORTED) {
    ESP_LOGW(TAG, "eFuse not burnt, skip software calibration");
} else {
    ESP_LOGE(TAG, "Invalid arg or no memory");
}
```

在获取 ADC 单次转换数据的过程中，有两种高效且灵活的方案。这两种方案均能够满足 ADC 单次转换数据获取的需求，用户可以根据自身项目的具体需求选择合适的方案。

- 方案一：见代码 3.7，首先调用 adc_oneshot_read()函数直接读取 ADC 单次转换的原始数据。然后使用 adc_cali_raw_to_voltage()函数将这些原始数据转换成校准后的

电压值（以 mV 为单位）。这种方案可以直接访问原始数据，并且允许用户根据需要进行后续的校准处理。

- 方案二：见代码 3.8，ESP-IDF 提供了一个更为便捷的函数 adc_oneshot_get_calibrated_result()，它作为一个快捷组合函数，能够一步到位地直接返回校准后的电压数据（以 mV 为单位）。在这个函数中，实际上已经隐含地执行了 adc_oneshot_read()函数和 adc_cali_raw_to_voltage()函数的操作，从而简化了用户的操作流程，提高了数据获取的效率。

代码 3.7　ADC单次转换获取数据的方案一

```
adc_oneshot_read(adc1_handle, ADC_CHANNEL_2, &adc_raw);
adc_cali_raw_to_voltage(adc1_cali_handle, adc_raw, &voltage);
```

代码 3.8　ADC单次转换获取数据的方案二

```
adc_oneshot_get_calibrated_result(adc1_handle, adc1_cali_handle,
ADC_CHANNEL_2, &voltage);
```

在结束 ADC 数据获取的操作后，为了确保资源有效利用以及系统的稳定性，务必记得回收 ADC 单元句柄和相关资源。可以使用相应的函数来释放 ADC 单元所占用的资源，见代码 3.9。

代码 3.9　ADC单元回收的关键代码

```
// 删除 ADC 单次转换的句柄
adc_oneshot_del_unit(adc1_handle);
// 删除 ADC 校准曲线拟合方案句柄
adc_cali_delete_scheme_curve_fitting(adc1_cali_handle);
```

3.3　RTC 应用

本节主要介绍 RTC 的基础知识和常用函数，然后通过一个动手实践项目让读者掌握 RTC 读取和设置的相关知识点。

3.3.1　RTC 简介

RTC（Real Time Clock，实时时钟）是现代集成电路中必备的模块，为系统提供稳定的时间基准。ESP32-C3 集成了一个 RTC 定时器，具有高精度、低功耗、睡眠模式也能保持系统时间等特点。不过，值得注意的是，ESP32-C3 上电复位时会导致 RTC 定时器重置，所以在实际应用中，为了确保时间准确，建议结合 NTP（Network Time Protocol，网络时间协议）使用。

3.3.2　RTC 的常用函数

RTC 的常用函数如表 3.8 所示，使用标准 C 函数或者 POSIX（Portable Operating System Interface）函数可以设置和获取当前时间。

表 3.8　RTC的常用函数

属性/函数	说　　明
time()	返回从1970年1月1日到当前时间的秒数
localtime()	将从1970年1月1日到当前时间的秒数转换成时间/日期结构
mktime()	将时间/日期结构转换成从1970年1月1日到当前时间的秒数
settimeofday()	设置时间和时区
gettimeofday()	获取时间和时区

其中有两个关键数据类型需要了解：

- time_t：从 1970 年 1 月 1 日到当前时间的秒数，见代码 3.10。
- struct tm：用来保存时间、日期和时区的结构体，见代码 3.11。

代码 3.10　从 1970 年 1 月 1 日到当前时间的秒数

```
typedef _TIME_T_    time_t;
#define _TIME_T_ long
```

代码 3.11　用来保存时间、日期和时区的结构体tm

```
struct tm
{
  int   tm_sec;
  int   tm_min;
  int   tm_hour;
  int   tm_mday;
  int   tm_mon;
  int   tm_year;
  int   tm_wday;
  int   tm_yday;
  int   tm_isdst;
#ifdef __TM_GMTOFF
  long  __TM_GMTOFF;
#endif
#ifdef __TM_ZONE
  const char *__TM_ZONE;
#endif
};
```

最后来看一下 time_t 和 struct tm 这两个关键数据类型的转换方法，如图 3.11 所示，time_t 通过 localtime()函数转换成 struct tm，struct tm 通过 mktime()函数转换成 time_t。

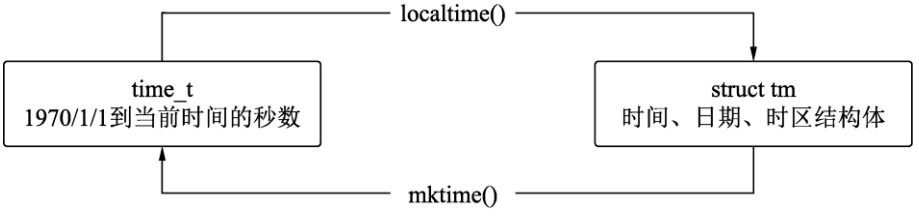

图 3.11　RTC 关键数据类型转换方法

3.3.3　实践：设置和获取 RTC 时间

【ESP32 源码路径：tutorial-esp32c3-getting-started/tree/master/peripheral/rtc】

本节主要利用前面介绍的 RTC 知识点和 RTC 的函数进行实践：读取和设置 RTC 系统时间。

1．操作步骤

（1）准备一个 ESP32-C3 开发板，通过 Visual Studio Code 开发工具编译 button 工程源码，生成相应的固件，再将固件下载到 ESP32-C3 开发板上。

（2）ESP32-C3 运行程序后，主程序每秒打印一次当前的时间，按 BOOT 键，设置 2024/01/10 00:00:00 时间到系统时间，程序运行结果如图 3.12 所示。

```
I (197) cpu_start: ESP-IDF:          v5.1.2-dirty
I (203) cpu_start: Min chip rev:     v0.3
I (207) cpu_start: Max chip rev:     v0.99
I (212) cpu_start: Chip rev:         v0.3
I (217) heap_init: Initializing. RAM available for dynamic allocation:
I (224) heap_init: At 3FC8CA50 len 000335B0 (205 KiB): DRAM
I (231) heap_init: At 3FCC0000 len 0001C710 (113 KiB): DRAM/RETENTION
I (238) heap_init: At 3FCDC710 len 00002950 (10 KiB): DRAM/RETENTION/STACK
I (245) heap_init: At 50000010 len 00001FD8 (7 KiB): RTCRAM
I (252) spi_flash: detected chip: generic
I (256) spi_flash: flash io: dio
I (260) sleep: Configure to isolate all GPIO pins in sleep state
I (267) sleep: Enable automatic switching of GPIO sleep configuration
I (274) app_start: Starting scheduler on CPU0
I (00:00:00.107) main_task: Started on CPU0
I (00:00:00.112) main_task: Calling app_main()
I (00:00:00.117) gpio: GPIO[8]| InputEn: 1| OutputEn: 0| OpenDrain: 0| Pullup: 1| Pulldown: 0| Intr:1
I (00:00:00.127) gpio: GPIO[9]| InputEn: 1| OutputEn: 0| OpenDrain: 0| Pullup: 1| Pulldown: 0| Intr:1
I (00:00:00.137) rtc: The current time is: 1970/01/01 00:00:00
I (00:00:01.137) rtc: The current time is: 1970/01/01 00:00:01
I (00:00:02.137) rtc: The current time is: 1970/01/01 00:00:02
I (00:00:03.137) rtc: The current time is: 1970/01/01 00:00:03
I (00:00:03.617) rtc: GPIO[9] 触发中断，当前电平：1
I (00:00:00.519) rtc: The current time is: 2024/01/10 00:00:00
I (00:00:01.519) rtc: The current time is: 2024/01/10 00:00:01
I (00:00:02.519) rtc: The current time is: 2024/01/10 00:00:02
I (00:00:03.519) rtc: The current time is: 2024/01/10 00:00:03
I (00:00:04.519) rtc: The current time is: 2024/01/10 00:00:04
```

图 3.12　读取和设置 RTC 系统时间的程序运行日志

2．程序源码解析

本次实践工程的源码结构如图 3.13 所示，包含 rtc.c、rtc.h、button.c、button.h 和 main.c 几个源文件。由于增加了 rtc.c 和 button.c 两个源文件，所以需要在 CMakeLists.txt 配置文件中通过 idf_component_register 将源文件添加到构建系统中。

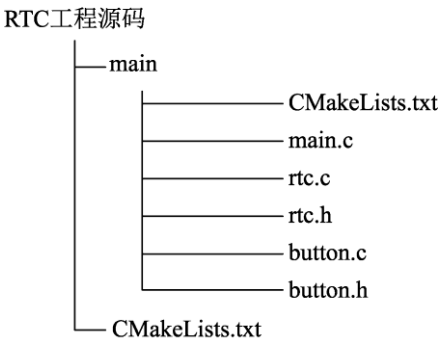

图 3.13　读取和设置 RTC 系统时间的工程源码结构

- main.c：程序流程如图 3.14 所示，app_main()函数是应用程序的入口，app_main()函数主要完成按键初始化和创建 rtcTask 任务。
  - app_main()函数随后进入一个循环，堵塞读取 button_queue 队列数据，等待按键按下触发中断向队列写入数据，一旦队列存在数据，就将 2024/01/10 00:00:00 设置为当前的系统时间。
  - rtcTask 任务也进入一个循环，以 1s 的间隔持续打印当前的系统时间。
- rtc.c 和 rtc.h：声明并设置系统时间（见代码 3.12）然后读取系统时间（见代码 3.13）。
- button.c 和 button.h：实现按键初始化函数和按键触发中断事件处理函数，详情可查阅 3.1.3 节。

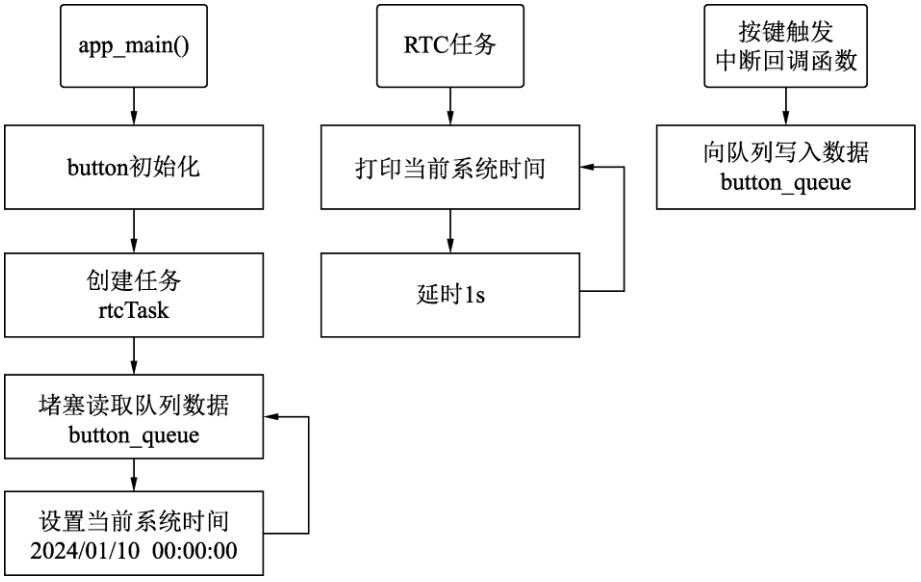

图 3.14　读取和设置 RTC 系统时间的程序流程

代码 3.12　RTC设置系统时间

```
/**
 * @brief 设置系统时间
 *
 * @param[struct tm] 设置的时间参数
 */
void set_time(struct tm datetime)
{
    time_t second = mktime(&datetime);
    struct timeval val = { .tv_sec = second, .tv_usec = 0 };
    settimeofday(&val, NULL);
}
```

代码 3.13　RTC读取系统时间

```
/**
 * @brief 读取当前系统时间
 *
 * @param[struct tm*] 读取的系统时间
 */
void get_time(struct tm* datetime)
```

```
{
    struct tm* temp;
    time_t second;
    time(&second);
    temp = localtime(&second);

    datetime->tm_sec = temp->tm_sec;
    datetime->tm_min = temp->tm_min;
    datetime->tm_hour = temp->tm_hour;
    datetime->tm_mday = temp->tm_mday;
    datetime->tm_mon = temp->tm_mon;
    datetime->tm_year = temp->tm_year;
    datetime->tm_wday = temp->tm_wday;
    datetime->tm_yday = temp->tm_yday;
    datetime->tm_isdst = temp->tm_isdst;
}
```

3.4　UART 通信

本节主要介绍 UART 的基础知识及其常用函数，并通过一个实践项目帮助读者掌握 UART 串口通信的相关知识点。

3.4.1　UART 简介

UART（Universal Asynchronous Receiver/Transmitter，通用异步收发器）是实现不同设备之间短距离通信的一种常用且经济的方式，最简单的连接方式如图 3.15 所示，只需要 3 根线，分别是串行输出线（TX）、串行输入线（RX）、地线（GND）。

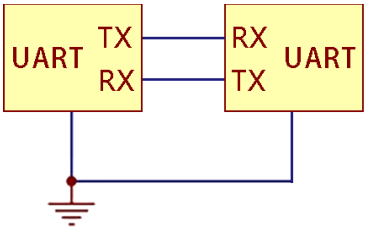

图 3.15　UART 串行通信连接方式

ESP32-C3 芯片集成了两个 UART，每个 UART 都可以独立配置波特率、数据位长度、位顺序、停止位、奇偶校验位等参数。

3.4.2　UART 的常用函数

UART 的常用函数如表 3.9 所示，其中 uart_param_config()是实现 UART 串口通信功能不可或缺的核心配置函数，该函数的入参和返回值如下：

```
/**
 * @brief 设置 UART 参数。
 *
 * @param[uart_port_t] uart_num: UART 端口号。
 * @param[uart_config_t *] uart_config: UART 配置参数。
 *
 * @return
 *      - ESP_OK，成功。
 *      - ESP_FAIL，失败。
```

```
 */
esp_err_t uart_param_config(uart_port_t uart_num, const uart_config_t
*uart_config);
```

表 3.9　UART的常用函数

属性/函数	说　　明
uart_driver_install()	为UART驱动程序分配ESP32-C3资源
uart_param_config()	设置UART参数（波特率、数据位、停止位等）
uart_set_pin()	设置UART通信的管脚
uart_write_bytes()	UART发送数据
uart_flush_input()	清空UART环形接收缓冲区的数据
uart_read_bytes()	UART接收数据

3.4.3　实践：通过 UART 串口与计算机通信

【ESP32 源码路径：tutorial-esp32c3-getting-started/tree/master/peripheral/uart】

本节主要利用前面介绍的 UART 知识点和 UART 的常用函数进行实践：通过 UART 串口实现 ESP32-C3 与计算机之间的通信。

1．操作步骤

（1）准备一个 ESP32-C3 开发板，按照表 3.10 所示做好硬件准备，并确保 ESP32-C3 与计算机正确连接。

表 3.10　USB转UART模块接线说明

外围设备引脚	ESP32-C3 开发板引脚	说　　明
VCC	VCC	电源正极3.3~5V
GND	GND	电源负极（地）
TX	RX(GPIO5)	模块TX接ESP32-C3的RX
RX	TX(GPIO4)	模块RX接ESP32-C3的TX

（2）通过 Visual Studio Code 开发工具编译 uart 工程源码，生成相应的固件，再将固件下载到 ESP32-C3 开发板上。

（3）在计算机上打开“串口调试工具”，按 ESP32-C3 开发板上的 BOOT 键，会触发串口发送数据 hello，在“串口调试工具”上将会显示“hello”数据。

（4）通过串口调试工具发送数据给 ESP32-C3，ESP32-C3 会将接收到的串口信息打印出来，程序运行日志如图 3.16 所示。

2．USB转UART模块接线说明

由于计算机没有 UART 接口，因此需要借助 USB 转 UART 模块进行信号中转。具体连接方式如表 3.10 所示。USB 转 UART 模块的 TX 端连接 ESP32-C3 开发板的 RX 端，USB 转 UART 模块的 RX 端接 ESP32-C3 开发板的 TX 端。

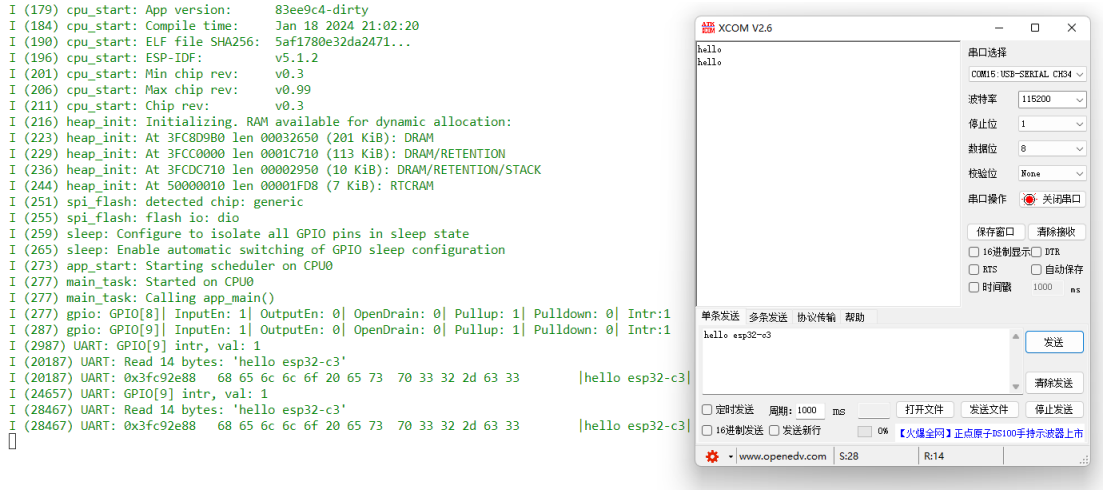

图 3.16　通过 UART 串口实现与计算机通信的程序运行日志

3．程序源码解析

本次实践工程的源码结构如图 3.17 所示，包含 uart.c、uart.h、button.c、button.h 和 main.c。注意，由于添加了 rmt_ws2812.c 源文件，为确保项目的正确编译和链接，需要在 CMakeLists.txt 配置文件中进行相应的配置。

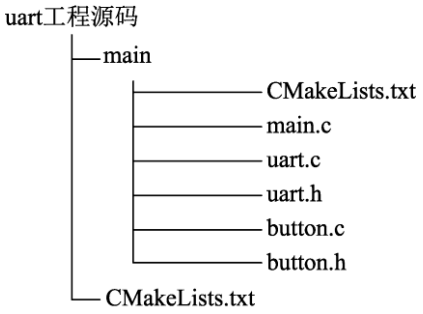

图 3.17　通过 UART 串口实现与计算机通信的工程源码结构

- main.c：程序流程如图 3.18 所示，在应用程序的入口 app_main()函数中完成按键初始化、串口初始化和创建 uartRxTask 任务。
  - app_main()函数最后进入一个循环，堵塞读取 button_queue 队列数据，等待按键按下后触发中断向队列写入数据。一旦队列存在数据，就将数据读取出来并转换成 GPIO 引脚号，然后判断如果是 BOOT 按键触发，则串口发送数据 hello。关键代码如代码 3.14 所示。
  - uartRxTask 任务也进入一个循环，堵塞接收串口数据，如果接收到串口数据，则通过日志将接收的数据打印出来。关键代码如代码 3.15 所示。
- uart.c 和 uart.h：声明并实现 UART 初始化函数，关键代码如代码 3.16 所示。
- button.c 和 button.h：实现按键初始化函数和按键触发中断事件处理函数，详情可查阅 3.1.3 节。

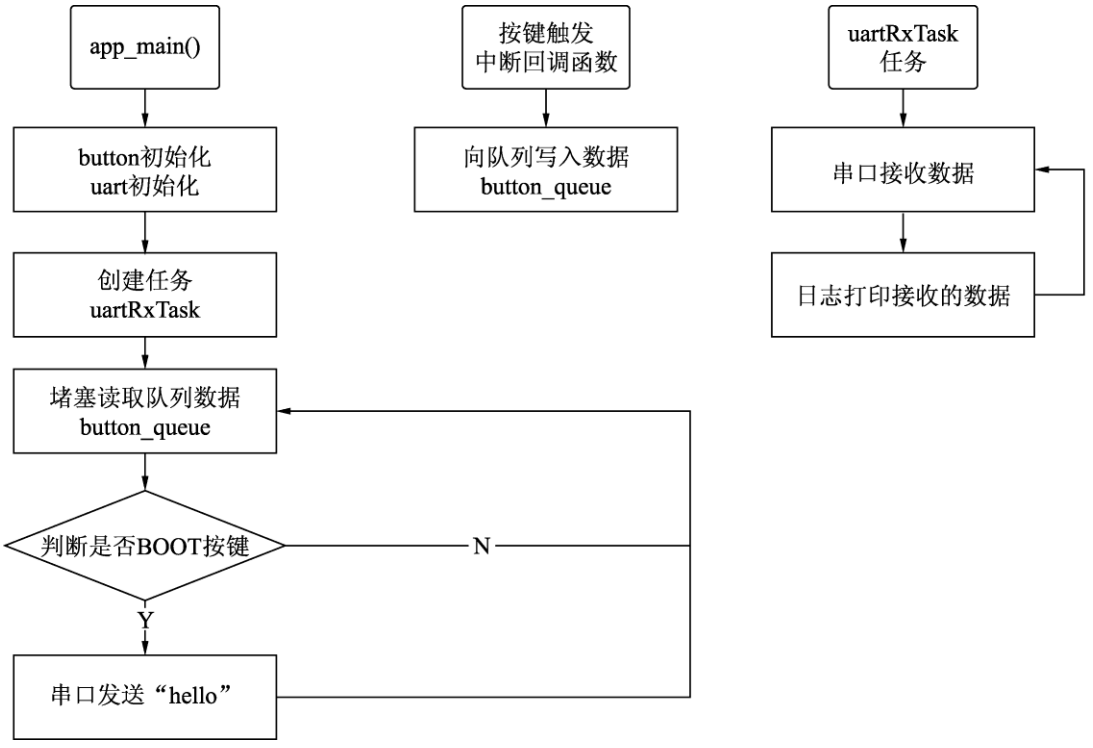

图 3.18　通过 UART 串口实现与计算机通信的程序流程

代码 3.14　通过UART串口实现与计算机通信主循环程序的关键代码

```
void app_main(void)
{
    uint32_t io_num;

    //按键初始化
    button_init();

    //串口初始化
    uart_init();

    //创建串口数据接收任务，堆栈大小为 4096，最大优先级
    xTaskCreate(uartRxTask, "uartRxTask", 4096, NULL, configMAX_PRIORITIES,
NULL);

    while(true) {
        // 堵塞读取队列中的数据
        if(xQueueReceive(button_queue, &io_num, portMAX_DELAY)) {
            // 如果读取成功，则打印出 GPIO 引脚号和当前的电平
            ESP_LOGI(TAG, "GPIO[%ld] 触发中断, 当前电平: %d", io_num,
gpio_get_level(io_num));
            if(io_num==GPIO_INPUT_BOOT) {
                uart_write_bytes(UART_NUM_1, "hello\r\n",
strlen("hello\r\n"));
            }
        }
    }
}
```

代码 3.15　通过UART数据接收任务的关键代码

```
/**
 * @brief UART 数据接收任务
 */
static void uartRxTask(void *arg)
{
    uint8_t* data = (uint8_t*) malloc(2048);
    while (1) {
        const int rxBytes = uart_read_bytes(UART_NUM_1, data, 2048,
pdMS_TO_TICKS(1000));
        if (rxBytes > 0) {
            data[rxBytes] = 0;
            ESP_LOGI(TAG, "Read %d bytes: '%s'", rxBytes, data);
            ESP_LOG_BUFFER_HEXDUMP(TAG, data, rxBytes, ESP_LOG_INFO);
        }
    }
    free(data);
}
```

代码 3.16　UART初始化函数

```
/**
 * @brief uart 初始化
 */
void uart_init(void) {
    // UART 配置参数
    const uart_config_t uart_config = {
        // 波特率
        .baud_rate = 115200,
        // 数据位
        .data_bits = UART_DATA_8_BITS,
        // 奇偶校验
        .parity = UART_PARITY_DISABLE,
        // 停止位
        .stop_bits = UART_STOP_BITS_1,
        // 硬件流控模式
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        // 时钟选择
        .source_clk = UART_SCLK_DEFAULT,
    };
    // 接口缓冲区 2048，发送缓冲区 0，事件队列 0
    uart_driver_install(UART_NUM_1, 2048, 0, 0, NULL, 0);
    // 设置 UART 参数
    uart_param_config(UART_NUM_1, &uart_config);
    // 设置 UART 硬件引脚
    uart_set_pin(UART_NUM_1, TXD_PIN, RXD_PIN, UART_PIN_NO_CHANGE,
UART_PIN_NO_CHANGE);
}
```

3.5　I^2C 通信

本节介绍 I^2C 的基础知识和 I^2C 的常用函数，然后通过一个动手实践项目让读者掌握 I^2C 通信的相关知识点。

3.5.1　I²C 简介

I²C（Inter-Integrated Circuit，集成电路总线）是一种近距离、串行、同步、多设备、半双工的两线制串行总线，由荷兰 PHILIPS 公司发明。如图 3.19 所示，I²C 允许一个主设备和多个从设备在同一个总线上共存。I²C 仅使用串行数据线（SDA）和串行时钟线（SCL）两条通信线。

ESP32-C3 集成了一个 I²C 控制器，既可以作为主控制器，也可以作为从控制器。ESP32-C3 同时支持 I²C 标准模式和快速模式，可分别达到 100kHz 和 400kHz 的通信速率。

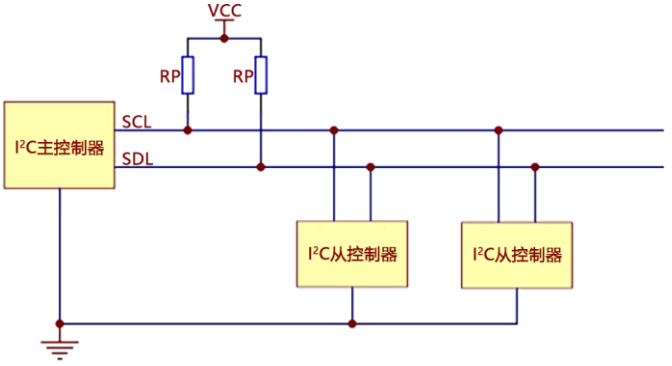

图 3.19　I²C 总线通信

3.5.2　I²C 的常用函数

I²C 的常用函数如表 3.11 所示，其中，i2c_param_config()是实现 I²C 通信不可或缺的核心配置函数，该函数的入参和返回值如下：

```
/**
 * @brief 配置 I2C 参数。
 *
 * @param[i2c_port_t] i2c_num: I2C 端口号。
 * @param[i2c_config_t *] i2c_conf: I2C 配置参数。
 *
 * @return
 *      - ESP_OK，成功。
 *      - ESP_FAIL，失败。
 */
esp_err_t i2c_param_config(i2c_port_t i2c_num, const i2c_config_t *i2c_conf);
```

表 3.11　I²C的常用函数

属性/函数	说　明
i2c_driver_install()	安装I²C总线驱动
i2c_param_config()	设置I²C总线参数
i2c_master_write_to_device()	对连接到I²C总线的设备执行写操作
i2c_master_write_read_device()	对连接到I²C总线的设备执行读操作
i2c_driver_delete()	删除I²C总线驱动

3.5.3　实践：通过 I²C 接口实现温度和湿度检测

【ESP32 源码路径：tutorial-esp32c3-getting-started/tree/master/peripheral/i2c】

本节主要利用前面介绍的 I²C 知识点和 I²C 的常用函数进行实践：通过 I²C 接口读取 AHT10 传感器的温度和湿度数据。

1．操作步骤

（1）准备一个 ESP32-C3 开发板，按照表 3.12 所示做好硬件准备。

表 3.12　AHT10 温度和湿度模块接线说明

外围设备引脚	ESP32-C3 开发板引脚	说　　明
VCC	VCC	电源正极3.3~5V
GND	GND	电源负极（地）
SCL	GPIO5	串行时钟线
SDA	GPIO4	串行数据线

（2）通过 Visual Studio Code 开发工具编译 i2c 工程源码，生成相应的固件，再将固件下载到 ESP32-C3 开发板上。

（3）ESP32-C3 运行程序后，在 I²C 初始化之后，会不断读取 AHT10 传感器的温度和湿度数据，这时候如果对 AHT10 哈口气，则温度和湿度数据飙升，片刻后才慢慢下降，这说明 AHT10 对环境的温度和湿度的灵敏度很高。程序运行日志如图 3.20 所示。

```
I (199) cpu_start: ESP-IDF:          v5.1.2
I (204) cpu_start: Min chip rev:     v0.3
I (209) cpu_start: Max chip rev:     v0.99
I (213) cpu_start: Chip rev:         v0.3
I (218) heap_init: Initializing. RAM available for dynamic allocation:
I (225) heap_init: At 3FC8DDC0 len 00032240 (200 KiB): DRAM
I (232) heap_init: At 3FCC0000 len 0001C710 (113 KiB): DRAM/RETENTION
I (239) heap_init: At 3FCDC710 len 00002950 (10 KiB): DRAM/RETENTION/STACK
I (246) heap_init: At 50000010 len 00001FD8 (7 KiB): RTCRAM
I (253) spi_flash: detected chip: generic
I (257) spi_flash: flash io: dio
I (261) sleep: Configure to isolate all GPIO pins in sleep state
I (268) sleep: Enable automatic switching of GPIO sleep configuration
I (275) app_start: Starting scheduler on CPU0
I (280) main_task: Started on CPU0
I (280) main_task: Calling app_main()
I (3880) I2C: 温度 = 23.301697 ℃, 湿度 = 77.439880 %
I (6880) I2C: 温度 = 23.301697 ℃, 湿度 = 77.439880 %
I (9880) I2C: 温度 = 23.222542 ℃, 湿度 = 76.603699 %
I (12880) I2C: 温度 = 26.470566 ℃, 湿度 = 99.998474 %
I (15880) I2C: 温度 = 26.788139 ℃, 湿度 = 99.998474 %
I (18880) I2C: 温度 = 26.916122 ℃, 湿度 = 99.998474 %
I (21880) I2C: 温度 = 26.911163 ℃, 湿度 = 99.998474 %
I (24880) I2C: 温度 = 26.963806 ℃, 湿度 = 99.998474 %
I (27880) I2C: 温度 = 27.001762 ℃, 湿度 = 99.998474 %
I (30880) I2C: 温度 = 27.488899 ℃, 湿度 = 99.998474 %
I (33880) I2C: 温度 = 28.153992 ℃, 湿度 = 84.727478 %
I (36880) I2C: 温度 = 27.796364 ℃, 湿度 = 69.432068 %
```

图 3.20　通过 I²C 接口实现温度和湿度检测的程序运行日志

2．温度和湿度传感器接线

温度和湿度传感器采用 AHT10 芯片，如图 3.21 所示。AHT10 是一款经过出厂标定校

准的高精度的贴片封装的温度和湿度传感芯片，集成了一个改进型的 MEMS 半导体电容式湿度传感器和一个标准的片上温度传感器原件，适用于空调和除湿器等温度和湿度控制领域的检测。

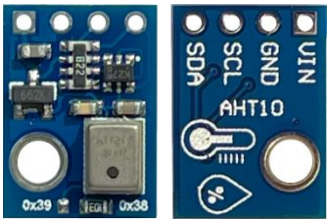

图 3.21　AHT10 温度和湿度传感器模块

AHT10 的工作特性如下：

- ❑ 工作电压：1.8～3.6V；
- ❑ 工作温度：−40℃～80℃；
- ❑ 湿度精度：±2%（典型值）；
- ❑ 湿度分辨率：0.024%（典型值）；
- ❑ 温度精度：±0.3℃（典型值）；
- ❑ 温度分辨率：0.01℃（典型值）；
- ❑ 接口类型：I^2C；
- ❑ 接线说明：如表 3.12 所示，ESP32-C3 作为 I^2C 主机，AHT10 作为 I^2C 从机。

3.程序源码解析

本次实践的程序流程如图 3.22 所示，从 app_main()入口函数开始，完成 I^2C 初始化、AHT10 开机和 AHT10 校准。最后程序进入一个循环，间隔 3s 循环通过 I^2C 读取 AHT10 原始数据，再将原始数据转换成温度和湿度，见代码 3.17。

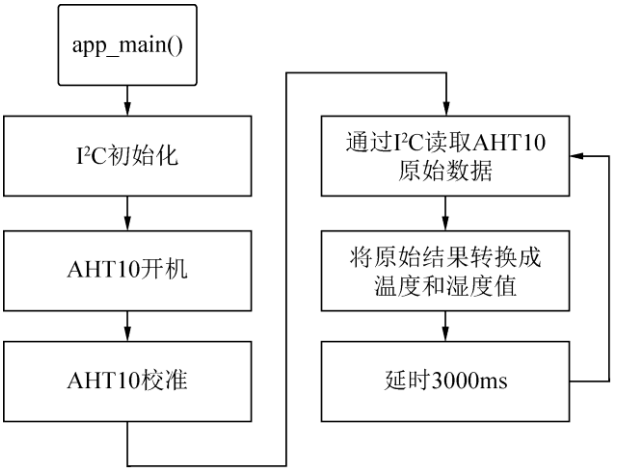

图 3.22　通过 I^2C 接口实现温度和湿度检测的程序流程

ESP32 通过 I^2C 接口与 AHT10 温度和湿度模块进行通信时，首先需要正确初始化 I^2C 接口。初始化过程涉及配置一系列关键参数，这些参数包括 I^2C 的工作模式、SDA（串行数据）引脚、SCL（串行时钟）引脚及时钟频率等。I^2C 初始化函数参见代码 3.17。

代码 3.17　I^2C初始化函数

```
/**
 * @brief I2C 初始化
 */
esp_err_t i2c_init(void)
{
```

```
    // I²C 配置参数
    i2c_config_t conf = {
        // I²C 模式，主机模式或者从机模式
        .mode = I2C_MODE_MASTER,
        // SDA 引脚
        .sda_io_num = 4,
        // SCL 引脚
        .scl_io_num = 5,
        // SDA 引脚上拉使能
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        // SCL 引脚上拉使能
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        // I²C 主机模式下的时钟频率
        .master.clk_speed = 400000,
    };

    // 设置 I²C 总线参数
    i2c_param_config(I2C_MASTER_NUM, &conf);
    // 安装 I²C 总线驱动
    return i2c_driver_install(I2C_MASTER_NUM, conf.mode, 0, 0, 0);
}
```

ESP32 完成 I²C 接口初始化后，即可通过 I²C 与 AHT10 温度和湿度模块建立通信。首先向 AHT10 写入开机命令，并等待其正常开机。接着再向 AHT10 写入校准命令，等待其校准完成。最后进入循环过程，不断地向 AHT10 写入读取数据命令，然后从 AHT10 中读取温度和湿度原始数据，具体实现可参考代码 3.18。

代码 3.18　通过I²C接口读写AHT10 温度和湿度模块的关键代码

```
// 对 AHT10 写入开机命令
aht10_register_write_byte(AHT10_CMD_NORMAL, data, 2);
vTaskDelay(pdMS_TO_TICKS(300));

//  对 AHT10 写入校准命令
aht10_register_write_byte(AHT10_CMD_CALIBRATION, data, 2);
vTaskDelay(pdMS_TO_TICKS(300));

while (1) {
    // 对 AHT10 写入读取数据命令
    aht10_register_write_byte(AHT10_CMD_GET_DATA, data, 2);

    // 延时 3000ms
    vTaskDelay(pdMS_TO_TICKS(3000));

    // 从 AHT10 中读取原始数据
    aht10_register_read(AHT10_CMD_GET_DATA, data, 6);

    // 将原始数据转换成温度和湿度数据
    temperature = ((data[3] & 0x0F) << 16 | data[4] << 8 | data[5]) * 200.0
/ (1 << 20) - 50;
    humidity = ((data[1]) << 12 | data[2] << 4 | (data[3] & 0xF0)) * 100.0
/ (1 << 20);
    ESP_LOGI(TAG, "温度 = %f ℃ , 湿度 = %f %%", temperature, humidity);
}
```

3.6　SPI 通信

本节介绍 SPI 的基础知识及其常用函数，然后通过一个动手实践项目让读者掌握 SPI 通信的相关知识点。

3.6.1　SPI 简介

SPI（Serial Peripheral interface，串行外围设备接口）是一种近距离、同步、多设备、全双工的四线制串行总线，由美国 MOTORLA 公司发明。如图 3.23 所示，SPI 允许一个主设备和多个从设备在同一个总线上共存，SPI 使用四线通信线：串行时钟线（SCLK）、主机输入/从机输出数据线（MISO）、主机输出/从机输入数据线（MOSI）和从机选择线（CS）。SPI 与 I^2C 的显著差异表现在以下几个方面：

- 数据线：SPI 的数据线有两根（MISO 和 MOSI），而 I^2C 的数据线仅有一根（SDA），所以 SPI 支持全双工通信，而 I^2C 只能支持半双工通信。
- 从机选择线：SPI 配备了从机选择线（CS），而 I^2C 则没有。SPI 通过控制 CS 线（低电平有效），可以快速确定跟哪个从机进行通信。而 I^2C 主机需要通过寻址，才能确定跟哪个从机通信。故相比之下，SPI 更加便捷和高速。

ESP32-C3 集成了 3 个 SPI 控制器，其中，SPI0 和 SPI1 控制器主要供内部访问 Flash 使用。因此对于常规用途，我们一般只能使用 SPI2 控制器。SPI2 的特性如下：

- 模式支持：支持主机模式和从机模式。
- 传输模式：支持 CPU 控制的传输模式和 DMA 控制的传输模式。
- 时钟频率：在主机模式下，时钟频率可达 80MHz；在从机模式下，时钟频率可达 60MHz。
- 信号总线：配备一条独立的信号总线，该总线有 6 条从机选择线（CS）。

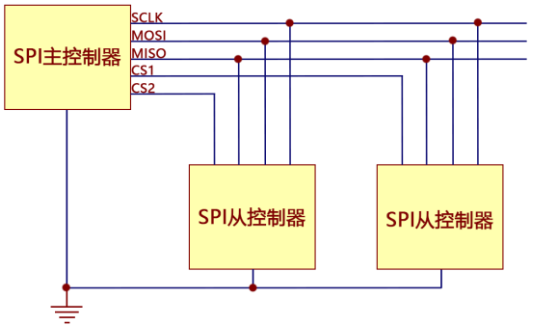

图 3.23　SPI 总线通信示意

3.6.2　SPI 的常用函数

SPI 的常用函数如表 3.13 所示。在通过 SPI 进行数据传输之前，需要在 SPI 总线上初

始化 SPI 从机设备，步骤如下：

（1）使用 spi_bus_initialize()函数初始化 SPI 总线上设备间共用的资源，如分配内存、初始化数据线 I/O、设置 DMA 和中断函数等。

（2）使用 spi_bus_add_device()函数将 SPI 添加设备到总线上，初始化该 SPI 设备的资源，如分配内存、初始化 CS I/O 等。

SPI 初始化完成之后，通过 SPI 进行数据传输有两种方式：

- 使用 spi_device_transmit()函数以中断形式传输数据。
- 使用 spi_device_polling_transmit()函数以轮询形式传输数据。

如果某个 SPI 设备的优先级较高，需要提高该 SPI 设备的传输效率，那么可以请求占用 SPI 总线，步骤如下：

（1）使用 spi_device_acquire_bus()函数请求占用 SPI 总线，以确保在该 SPI 设备使用过程中没有其他 SPI 设备干扰。

（2）使用 spi_device_release_bus()函数释放占用的 SPI 总线，使其重新可供其他设备使用。

完成 SPI 数据传输后，需要释放相关资源，以优化系统性能，步骤如下：

（1）使用 spi_bus_remove_device()函数在 SPI 总线上移除设备，并释放该 SPI 设备所占用的内存和资源。

（2）使用 spi_bus_free()函数释放 SPI 总线，移除总线上的所有设备，并释放 SPI 总线的所有资源。

表 3.13　SPI的常用函数

属性/函数	说　　明
spi_bus_initialize()	SPI总线初始化
spi_bus_free()	释放SPI总线，移除总线上的所有设备
spi_bus_add_device()	添加SPI总线上的设备
spi_bus_remove_device()	移除SPI总线上的设备
spi_device_transmit()	以中断形式传输数据
spi_device_polling_transmit()	以轮询方式传输数据
spi_device_acquire_bus()	请求占用SPI总线，使设备可以进行传输
spi_device_release_bus()	释放占用的SPI总线，使其他设备可以进行传输

3.6.3　实践：通过 SPI 接口实现外部存储模块的读写

【ESP32 源码路径：tutorial-esp32c3-getting-started/tree/master/peripheral/spi_w25qxx】

本节主要利用前面介绍的 SPI 知识点及其常用函数进行项目实践：通过 SPI 接口对 W25QXX 进行数据读写。

1．操作步骤

（1）准备一个 ESP32-C3 开发板，按照表 3.14 所示做好硬件准备。

（2）通过 Visual Studio Code 开发工具编译 spi_w25qxx 工程源码，生成相应的固件，

再将固件下载到 ESP32-C3 开发板上。

表 3.14　W25QXX存储模块接线说明

外围设备引脚	ESP32-C3 开发板引脚	说　　明
VCC	VCC	电源正极3.3~5V
GND	GND	电源负极（地）
MISO	GPIO2	主机输入/从机输出数据线
MOSI	GPIO7	主机输出/从机输入数据线
SCK	GPIO6	串行时钟线
CS	GPIO10	从机选择线

（3）ESP32-C3 运行程序后，在 SPI 初始化之后，向 W25Q16 写入“hello 小康师兄”数据，然后读取出来。程序运行日志如图 3.24 所示。

```
I (196) cpu_start: ESP-IDF:          v5.1.2-dirty
I (202) cpu_start: Min chip rev:     v0.3
I (206) cpu_start: Max chip rev:     v0.99
I (211) cpu_start: Chip rev:         v0.4
I (216) heap_init: Initializing. RAM available for dynamic allocation:
I (223) heap_init: At 3FC8F0A0 len 00030F60 (195 KiB): DRAM
I (229) heap_init: At 3FCC0000 len 0001C710 (113 KiB): DRAM/RETENTION
I (236) heap_init: At 3FCDC710 len 00002950 (10 KiB): DRAM/RETENTION/STACK
I (244) heap_init: At 50000010 len 00001FD8 (7 KiB): RTCRAM
I (251) spi_flash: detected chip: generic
I (255) spi_flash: flash io: dio
I (259) sleep: Configure to isolate all GPIO pins in sleep state
I (266) sleep: Enable automatic switching of GPIO sleep configuration
I (273) app_start: Starting scheduler on CPU0
I (278) main_task: Started on CPU0
I (278) main_task: Calling app_main()
I (278) gpio: GPIO[10]| InputEn: 0| OutputEn: 1| OpenDrain: 0| Pullup: 0| Pulldown: 0| Intr:0
I (288) SPI_W25QXX: id=0xEF14
I (588) SPI_W25QXX: W25QXX_Write: hello 小康师兄
I (588) SPI_W25QXX: W25QXX_Read : hello 小康师兄
```

图 3.24　通过 SPI 接口读写外部 Flash 存储模块的程序运行日志

2．外部Flash存储模块接线

外部 Flash 存储模块采用的是 W25QXX 芯片，如图 3.25 所示。W25Qxx 系列是一种低成本、小型化、使用简单的非易失性存储器，常应用于数据存储、字库存储、固件程序存储等场景。W25QXX 的工作特性如下：

图 3.25　W25QXX 存储模块

- ❑ 存储介质：Nor Flash（闪存）；
- ❑ 存储容量：见表 3.15；
- ❑ 工作电压：2.7～3.6V；
- ❑ 时钟频率：80MHz、160MHz（Dual SPI）、320MHz（Quad SPI）；
- ❑ 接口类型：SPI；
- ❑ 接线说明：如表 3.15 所示，ESP32-C3 作为 SPI 主机，W25QXX 作为 SPI 从机。

表 3.15　W25QXX型号与容量对应表

型　　号	容　　量
W25Q40	4Mbit / 512 KB
W25Q80	8Mbit / 1MB
W25Q16	16Mbit / 2MB
W25Q32	32Mbit / 4MB
W25Q64	64Mbit / 8MB
W25Q128	128Mbit / 16MB
W25Q256	256Mbit / 32MB

3．程序源码解析

本次实践的程序流程如图 3.26 所示，从 app_main()入口函数开始，首先初始化 W25QXX，然后向 W25QXX 写入数据，最后从 W25QXX 中读取数据，参见代码 3.19。

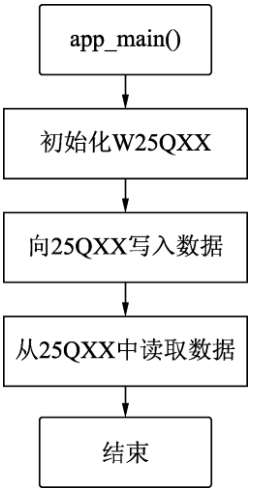

图 3.26　通过 SPI 接口读写 W25QXX 的程序流程

代码 3.19　SPI接口读写W25QXX主程序的关键代码

```
void app_main(void)
{
    uint8_t temp[256] = {0};
    uint8_t TEXT_Buffer[] = {"hello 小康师兄"};

    // 初始化 W25QXX
    W25QXX_Init();

    // 向 W25QXX 写入数据
    W25QXX_Write(TEXT_Buffer, 0, sizeof(TEXT_Buffer));
    ESP_LOGI(TAG, "W25QXX_Write: %s", TEXT_Buffer);

    // 从 W25QXX 中读取数据
    W25QXX_Read(temp, 0, sizeof(TEXT_Buffer));
    ESP_LOGI(TAG, "W25QXX_Read : %s", temp);
}
```

ESP32 通过 SPI 接口与 W25QXX 存储模块进行通信时，需要执行一系列初始化步骤。首先初始化 CS（片选）引脚，设置为 GPIO 输出模式。其次初始化 SPI 总线，确定 MISO、MOSI 和 SCLK 等引脚。然后初始化 SPI 设备并将其添加到 SPI 总线上，以便 ESP32 能够识别并与 W25QXX 存储模块建立通信。最后读取 W25QXX 模块型号，确保后续通信的兼容性。关键代码请参见代码 3.20。

代码 3.20　W25QXX初始化的关键代码

```
#define SPI_HOST            SPI2_HOST
#define PIN_NUM_CS          10
#define PIN_NUM_MISO        2
#define PIN_NUM_MOSI        7
#define PIN_NUM_CLK         6

#define W25QXX_CS_L         gpio_set_level(PIN_NUM_CS, 0);
#define W25QXX_CS_H         gpio_set_level(PIN_NUM_CS, 1);

/**
 * @brief W25QXX 初始化
 */
void W25QXX_Init(void)
{
    // CS 引脚 GPIO 输出初始化
    gpio_config_t io_conf;
    // 设置 I/O 方向为输出
    io_conf.mode = GPIO_MODE_OUTPUT;
    // 设置 I/O 引脚
    io_conf.pin_bit_mask = (1ULL << PIN_NUM_CS);
    //设置 I/O 上拉：使能
    io_conf.pull_up_en = 1;
    //配置 GPIO 参数
    gpio_config(&io_conf);

    // 取消片选
    W25QXX_CS_H;

    // SPI 总线配置参数
    spi_bus_config_t buscfg = {
        .miso_io_num = PIN_NUM_MISO,
        .mosi_io_num = PIN_NUM_MOSI,
        .sclk_io_num = PIN_NUM_CLK,
    };
    // SPI 设备接口参数
    spi_device_interface_config_t devcfg = {
        // 时钟频率
        .clock_speed_hz = 20 * 1000 * 1000,
        // SPI 模式(CPOL, CPHA)
        .mode = 0,
        // 传输队列大小
        .queue_size = 6,
    };

    // SPI 总线初始化
    spi_bus_initialize(SPI_HOST, &buscfg, 0);
    // 添加 SPI 总线设备
    spi_bus_add_device(SPI_HOST, &devcfg, &g_spi);
```

```
    //读取 FLASH ID
    uint16_t W25QXX_TYPE = 0;
    W25QXX_TYPE = W25QXX_ReadID();
    ESP_LOGI(TAG, "id=0x%X", W25QXX_TYPE);

    if(W25QXX_TYPE==W25Q256)
    {
        // 读取状态寄存器 3，判断地址模式
        uint8_t temp=W25QXX_ReadSR(3);
        // 如果不是 4 字节地址模式，则进入 4 字节地址模式
        if((temp&0X01)==0)
        {
            // 片选使能
            W25QXX_CS_L;
            // 发送进入 4 字节地址模式的指令
            SPI_ReadWriteByte(W25X_Enable4ByteAddr);
            // 取消片选
            W25QXX_CS_H;
        }
    }
}
```

封装一个 SPI 接口的读写函数，使用 spi_device_transmit()函数并通过中断方式在 SPI 总线上传输数据。该函数的关键在于配置 spi_transaction_t 结构体，指定发送和接收数据的缓冲区，每次仅传输一个字节。函数参数是需要发送的数据，函数返回值是接收到的数据。该函数简化了 SPI 通信的复杂性，使得后续通过 SPI 接口对 W25QXX 等设备进行读写操作变得更加便捷和高效，参见代码 3.21。

代码 3.21　通过SPI接口读写的关键函数

```
/**
 * @brief SPI 读写一个字节
 *
 * @param[uint8_t] TxData: 要写入的字节
 *
 * @return[uint8_t] 读取到的字节
 */
uint8_t SPI_ReadWriteByte(uint8_t TxData)
{
    uint8_t Rxdata;
    // SPI 传输事务结构体
    spi_transaction_t t;
    memset(&t, 0, sizeof(t));
    t.length = sizeof(uint8_t) * 8;
    t.rxlength = sizeof(uint8_t) * 8;
    t.tx_buffer = &TxData;
    t.rx_buffer = &Rxdata;
    // 以中断形式在 SPI 总线上传输数据
    spi_device_transmit(g_spi, &t);
    //返回收到的数据
    return Rxdata;
}
```

从 W25QXX 存储模块中读取数据相对简单、直接，无须额外的使能步骤或特殊操作，也没有页的限制。数据读取完成后，存储模块不会进入忙状态，但请确保在存储模块非忙状态时进行读取操作，以避免潜在的数据冲突。关键代码请参见代码 3.22。

代码 3.22　从W25QXX存储模块中读取数据的关键代码

```
/**
 * @brief 读取 SPI FLASH，从指定地址开始读取指定长度的数据
 *
 * @param[uint8_t *] pBuffer: 数据存储区
 * @param[uint32_t] ReadAddr: 开始读取的地址(24bit)
 * @param[uint16_t] NumByteToRead: 要读取的字节数(最大 65535)
 */
void W25QXX_Read(uint8_t *pBuffer, uint32_t ReadAddr, uint16_t
NumByteToRead)
{
    uint32_t i;
    // 片选使能
    W25QXX_CS_L;
    //发送读取命令
    SPI_ReadWriteByte(W25X_ReadData);
    //发送 24bit 地址
    SPI_ReadWriteByte((uint8_t)((ReadAddr) >> 16));
    SPI_ReadWriteByte((uint8_t)((ReadAddr) >> 8));
    SPI_ReadWriteByte((uint8_t)ReadAddr);
    for (i = 0; i < NumByteToRead; i++)
    {
        //循环读数
        pBuffer[i]=SPI_ReadWriteByte(0XFF);
    }
    // 取消片选
    W25QXX_CS_H;
}
```

向 W25QXX 存储模块写入数据时，为了确保数据的安全性和准确性，需要注意以下几点：

- 写使能（写保护解除）：在写入数据之前，先执行写使能操作。这是为了保护存储模块，避免因意外写入造成的数据损失。
- 擦除操作：由于 Flash 存储区的写入特性，每个数据位只能从 1 改写为 0，而不能从 0 改写为 1。因此，在写入新数据之前，必须对目标存储区域进行擦除操作，即将该区域的所有数据位设置为 1（即 0xFF），表示 Flash 中的空白状态。擦除操作必须按照最小的擦除单元（通常为 4KB 的扇区）进行，即使需要擦除的数据小于一个扇区，也必须对整个扇区进行擦除。
- 写入限制：当连续写入多字节数据时，每次最多只能写入一页数据（如 4096 字节）。如果写入的数据量超过一页，则会从页首开始覆盖写入，因此需要根据写入的地址和数据量分多次写入。
- 状态检查：写入操作完成后，存储芯片会进入忙状态，此时不响应新的读写请求。为了确定芯片是否已准备好以便进行后续操作，需要检查其状态寄存器。只有当状态寄存器的值为 0 时，才表示芯片已不处于忙状态，可以进行下一步操作。

W25QXX 存储模块写入数据的关键代码见代码 3.23。

代码 3.23　向W25QXX存储模块写入数据的关键代码

```
/**
 * @brief 写 SPI FLASH，从指定地址开始写入指定长度的数据
 *
```

```
 * @param[uint8_t *] pBuffer: 数据存储区
 * @param[uint32_t] WriteAddr: 开始写入的地址(24bit)
 * @param[uint16_t] NumByteToWrite: 要写入的字节数(最大 65535)
 */
uint8_t W25QXX_BUFFER[4096];
void W25QXX_Write(uint8_t *pBuffer, uint32_t WriteAddr, uint16_t
NumByteToWrite)
{
    uint32_t secpos;
    uint16_t secoff;
    uint16_t secremain;
    uint16_t i;
    // 扇区地址，一个扇区为 4096 字节
    secpos = WriteAddr / 4096;
    // 在扇区内的偏移
    secoff = WriteAddr % 4096;
    // 扇区剩余空间大小
    secremain = 4096 - secoff;

    if (NumByteToWrite <= secremain)
        secremain = NumByteToWrite;

    while (1) {
        // 读出整个扇区的内容
        W25QXX_Read(W25QXX_BUFFER, secpos * 4096, 4096);
        for (i = 0; i < secremain; i++) {
            // 校验数据
            if (W25QXX_BUFFER[secoff + i] != 0XFF)
                break;
        }
        if (i < secremain) {
            // 需要擦除这个扇区
            W25QXX_Erase_Sector(secpos);
            for (i = 0; i < secremain; i++) {
                // 复制补充需要写入的数据
                W25QXX_BUFFER[i + secoff] = pBuffer[i];
            }
            // 写入整个扇区
            W25QXX_Write_NoCheck(W25QXX_BUFFER, secpos * 4096, 4096);
        } else {
            // 不需要擦除，直接写入
            W25QXX_Write_NoCheck(pBuffer, WriteAddr, secremain);
        }
        if (NumByteToWrite == secremain) {
            // 写入结束
            break;
        } else {
            // 扇区地址增 1
            secpos++;
            // 偏移位置为 0
            secoff = 0;

            // 需要写入的数据存储+已写入的字节数
            pBuffer += secremain;
            // 写入地址+已写入的字节数
            WriteAddr += secremain;
            // 需要写入的字节数-已经写入的字节数
            NumByteToWrite -= secremain;
```

```
                if (NumByteToWrite > 4096) {
                    //下一个扇区还是写不完
                    secremain = 4096;
                } else {
                    //下一个扇区可以写完了
                    secremain = NumByteToWrite;
                }
            }
        }
    }
```

3.7　RMT 应用

本节介绍 RMT 的基础知识及其常用函数，然后通过一个动手实践项目帮助读者掌握 RMT 接口的相关知识点。

3.7.1　RMT 简介

RMT（Remote Control Transceiver，遥控收发器）通常被设计为红外发射器和红外接收器。由于其出色的灵活性，经常被进一步扩展成通用收发器，用于发送和接收多种类型的信号。

在发射器模式下，RMT 可以根据用户数据生成相应的波形。首先由 RMT 驱动程序将用户数据编码为 RMT 数据格式，随后由 RMT 发射器根据编码生成相应的波形，可以选择性将其调制上高频载波信号，以增强信号的抗干扰能力，最后将波形通过 GPIO 引脚发送出去。

在接收器模式下，RMT 能够对输入信号进行高精度采样，并支持从高频调制信号中解调出原始信号，再将其转换为 RMT 数据格式，最后将数据存储在内存中。此外，RMT 接收器还支持信号分析，能够识别信号的停止条件，并有效过滤掉噪声信号，确保接收数据的准确性和可靠性。

ESP32 的 RMT 外设具备多个独立通道，每个通道均可独立配置为发射器或接收器。这一特性使得 RMT 能够同时处理多个信号任务，大大提高了系统的并行处理能力。此外，RMT 还支持发送或接收任何使用 IR 协议的红外信号，如 NEC+协议，还可以作为通用序列发生器。

只要实现了适当的编码器，RMT 外围设备几乎可以生成任何波形。RMT 编码器用于将用户数据（如 RGB 像素）编码为可由硬件识别的格式。

3.7.2　RMT 的常用函数

在 ESP-IDF 5.x 版本中，RMT 驱动程序有较大的更新，在核心概念和主要使用方法上存在显著的变化。因为新版本的 RMT 驱动程序变化较大且使用更加复杂，而旧版本的 RMT 驱动程序仍然可用，所以在 driver/rmt.h 头文件路径中保留。本节将继续介绍和使用旧版本的 RMT 驱动程序，其常用函数如表 3.16 所示。

表 3.16　RMT旧版本中的常用函数

属性/函数	说　　明
rmt_config()	配置RMT参数
rmt_driver_install()	初始化RMT驱动程序
rmt_write_sample()	将整数数组的用户数据转换成RMT符号再发送出去
rmt_wait_tx_done()	等待RMT发送完成
rmt_translator_init()	初始化RMT编码器，并注册用户数据转为RMT符号的函数
rmt_driver_uninstall()	卸载RMT驱动程序

3.7.3　实践：通过 RMT 接口实现 RGB LED 灯带控制

【ESP32 源码路径：tutorial-esp32c3-getting-started/tree/master/peripheral/led_strip】

本节主要利用前面介绍的 RMT 知识点和常用函数进行项目实践：通过 RMT 接口控制 SK68XXMINI-HS　RGB LED 灯带的颜色亮灭。

1．操作步骤

（1）准备一个 ESP32-C3 开发板，通过 Visual Studio Code 开发工具编译 led_strip 工程源码，生成相应的固件，再将固件下载到 ESP32-C3 开发板上。

（2）ESP32-C3 运行程序后，灯带实物运行效果如图 3.27 所示，间隔 3s 轮流显示红色、绿色和蓝色。

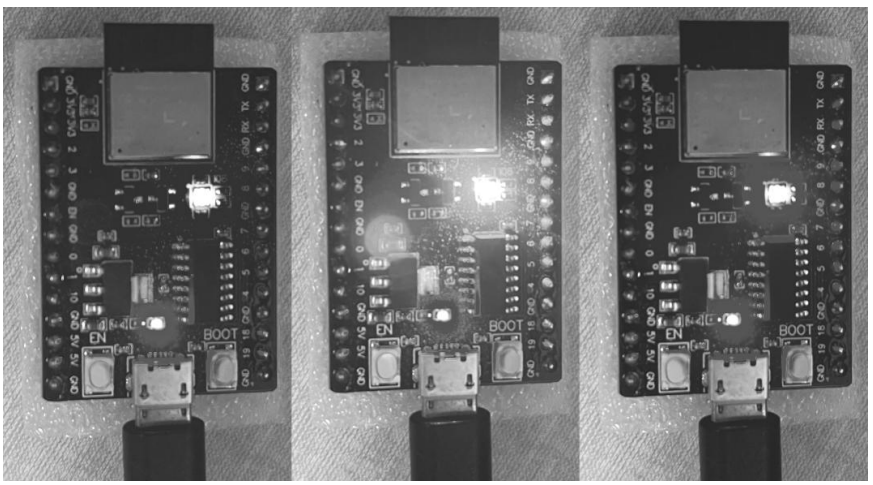

图 3.27　通过 RMT 接口实现彩色 LED 灯带控制的效果图

2．RGB LED接线

ESP32-C3 开发板上集成了 SK68XXMINI-HS 型号的 RGB LED，硬件原理如图 3.28 所示，通过一个 GPIO 就能实现对 RGB LED 颜色和亮度的精准控制。

SK68XXMINI 是一款集控制电路与发光电路于一体的智能外控 LED 光源。每个光源灯珠均包含一个像素点，每个像素点都能实现三基色颜色的 256 级亮度显示。LED 支持单

线数据传输的串行级联接口，仅需一根信号线就能完成数据的接收和解码，并且支持无限级联，极大地简化了布线工作。此外，LED 内置的信号整形电路确保了信号在经过每个像素点后都能保持稳定的波形，有效避免了波形畸变的累加，保证了信号的传输质量和稳定性。

在同类产品中，除了 SK68XXMINI 外，还有 WS2812B 可供选择，并且两者的用法完全一致。

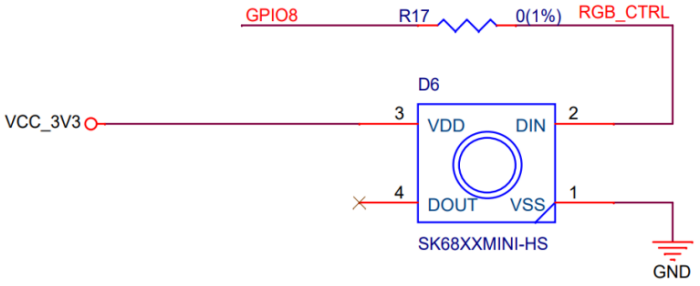

图 3.28　SK68XXMINI-HS 彩色 LED 的硬件原理

3．程序源码解析

本次实践工程的源码结构如图 3.29 所示，其核心源码包括 rmt_ws2812.c、rmt_ws2812.h 和 main.c 文件。为了实现对 RMT 驱动 WS2812 模块的深度封装，我们创建了 rmt_ws2812 源文件集，这不仅简化了 main.c 的复杂度，还显著提高了代码的可读性和可维护性。通过这个封装过程，成功地将 RMT 功能模块化，极大地简化了 RGB LED 控制流程。这使得开发者能够更加专注于业务逻辑的实现，无须深入了解底层硬件的细节。

注意，由于添加了 rmt_ws2812.c 源文件，为了确保项目的正确编译和链接，需要在 CMakeLists.txt 配置文件中进行相应的配置。

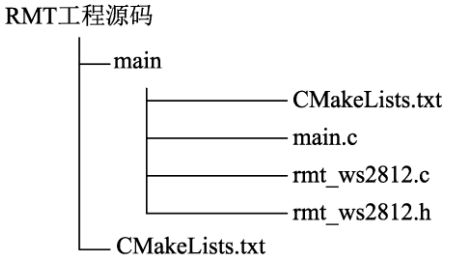

图 3.29　通过 RMT 接口实现 RGB LED 灯带控制的工程源码结构

- main.c：见代码 3.24，程序流程如图 3.30 所示，在应用程序的入口 app_main()函数中完成 WS2812 初始化，然后间隔 3s 循环控制 RGB LED 显示红色、绿色和蓝色。
- 在 rmt_ws2812.c 和 rmt_ws2812.h 中主要实现了如下几个函数。
  - ws2812_init()：初始化 RMT 驱动程序并创建 WS2812 结构体。
  - ws2812_deinit()：卸载 RMT 驱动程序，并释放 WS2812 结构体占用的内存资源。
  - ws2812_set_pixel()：设置 RGB 数值到 WS2812 结构体的内存。
  - ws2812_refresh()：刷新灯带上各个 LED 的颜色状态，将 WS2812 结构体内存中

的数据通过 RMT 发送到各个 LED 中。

- ws2812_clear()：关闭灯带上的所有 LED。

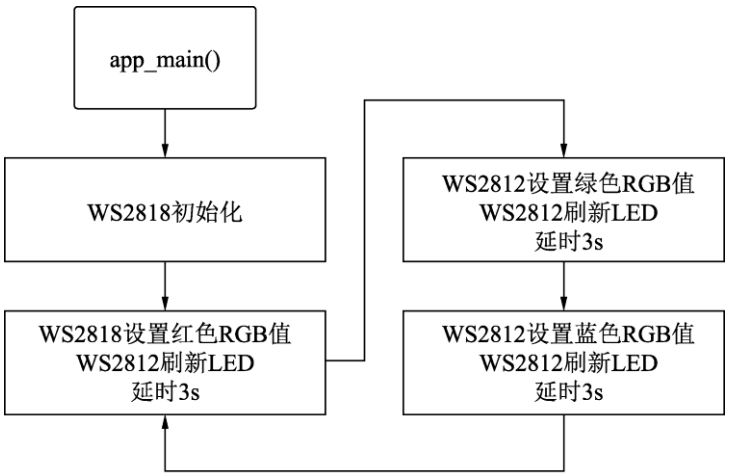

图 3.30　通过 RMT 接口实现 RGB LED 灯带控制的程序流程

代码 3.24　通过RMT接口实现RGB LED灯带控制的主程序代码

```
#include "esp_log.h"
#include "rmt_ws2812.h"

// RGB LED 灯带个数
#define     LED_STRIP_NUMBER            1
// RMT 发射信号引脚
#define     RMT_TX_GPIO                 8

// 全局常量字符串，用于日志打印的标签
const static char* TAG = "RMT";

void app_main(void)
{
    // 初始化 RMT 驱动程序，并创建 WS2812 结构体
    ws2812_t *ws2812 = ws2812_init(RMT_CHANNEL_0, RMT_TX_GPIO,
LED_STRIP_NUMBER);
    while (true) {
        // 在 WS2812 结构体的内存中设置红色的 RGB 数值
        ws2812_set_pixel(ws2812, 0, 255, 0, 0);
        // 通过 RMT 将 WS2812 结构体内存中的数据发送到各个 LED 中
        ws2812_refresh(ws2812, 50);
        // 延时 3s
        vTaskDelay(pdMS_TO_TICKS(3000));

        // 在 WS2812 结构体的内存中设置绿色的 RGB 数值
        ws2812_set_pixel(ws2812, 0, 0, 255, 0);
        //通过 RMT 将 WS2812 结构体内存中的数据发送到各个 LED 中

        ws2812_refresh(ws2812, 50);
        // 延时 3s
        vTaskDelay(pdMS_TO_TICKS(3000));

        // 设置绿色的 RGB 数值到 WS2812 结构体的内存中
```

```
            ws2812_set_pixel(ws2812, 0, 0, 0, 255);
            // 通过 RMT 将 WS2812 结构体内存中的数据发送到各个 LED 中
            ws2812_refresh(ws2812, 50);
            // 延时 3s
            vTaskDelay(pdMS_TO_TICKS(3000));
        }
    }
```